Appendix A

Scheduling using Behavioral Templates

Tai Ly, David Knapp, Ron Miller, Don MacMillen
Synopsis Inc.
700B E. Middlefield Road
Mountain View, CA USA 94043

Abstract: This paper presents the idea of "behavioral templates" in scheduling. A behavioral template locks several operations into a relative schedule with respect to one another. This simple construct proves powerful in addressing: (1) timing constraints, (2) sequential operation modeling, (3) pre-chaining of certain operations, and (4) hierarchical scheduling. We present design examples from industry to demonstrate the importance of these issues in scheduling.

1.0 Introduction

The task of scheduling [4] is to sequence nodes in a control and data flow graph (CDFG) by assigning each node to a control step (*cstep*). We present the idea of behavioral templates, and describe how we use behavioral templates to address several issues that arise when applying scheduling to commercial designs. For the purpose of this paper, we assume timing constrained scheduling [5].

A behavioral template specifies a relative scheduling among its member CDFG nodes. It is a template in the sense that its member nodes can be treated as a single scheduling unit by assigning the starting cstep for the template. It is behavioral in the sense that it specifies a behavioral pattern as opposed to, for example, a structural pattern [8]. We extend scheduling algorithms to handle behavioral templates by recasting the task of scheduling as that of assigning templates to csteps.

Although a simple idea, behavioral templates provide a powerful way to address four

issues in scheduling:

1. Timing constraints. We use behavioral templates to impose fixed and maximum timing constraints. This is more efficient than using precedence edges alone because an entire sequence of nodes is considered at once when scheduling one template.

2. Multi-cycle operations. To enable scheduling of complex multi-cycle operations, we use multiple CDFG nodes locked in a behavioral template to model the cycle-by-cycle I/O and resource requirements of such operations.

3. Logic and bit-manipulation operations. We use behavioral templates to force certain chaining of logic and bit-manipulation operations to save register costs. This reduces the scheduling design space, and therefore run times.

4. CDFG hierarchy. We implement hierarchical scheduling by inlining each scheduled subgraph, using a behavioral template to lock the inlined nodes according to the subgraph's schedule.

This paper is organized as follows. Section 2 compares this work to previous research. Section 3 defines behavioral templates. Section 4 describes extending scheduling for behavioral templates. Section 5 discusses applications. Section 6 presents results. Section 7 concludes this paper.

2.0 Related Work

The term "template" was used in [8] to describe structural patterns to exploit regularity. In [9] and [10], such templates are used to guide the clustering of CDFG nodes into super nodes which map to "regular" subcircuits. Both of these works focus on extracting regular patterns by pattern matching, whereas our work focuses on how to

schedule a set of behavioral patterns. Our behavioral templates do not represent *repeating* patterns, but specify local scheduling constraints among CDFG nodes.

Most scheduling systems model multi-cycle operations using single CDFG nodes whose delays are greater than 1. In [7], multi-cycle operations are treated as multiple single-cycle operations. This turns out to be similar to our template-based model for sequential operations, except that we make deliberate use of cycle-by-cycle input/output and resource requirements to model complex operations.

Hierarchical scheduling based on super nodes are used in [9], [6], and [7]. We know of no other system which hierarchically schedules a design while taking advantage of possible resource sharing between nodes and edges in different subgraphs.

3.0 Behavioral Templates

We define a behavioral template, T, as a CDFG object which specifies a set of tuples, (n_i, o_i), where n_i is a CDFG node and o_i is an integer cycle *offset*. The semantics is that T imposes the constraint:

$$\text{schedule}(n_i) = \text{schedule}(T) + o_i \qquad \text{for all } (n_i, o_i) \text{ in } T$$

where schedule(n_i) and schedule(T) denotes the schedules for n_i and T, respectively.

That is, if T is scheduled to cstep j, then every member node, n_i, of T must be scheduled to the cstep, $j + o_i$. This locks all nodes in T into a pattern of relative schedules, and we may schedule the entire group of nodes by scheduling the template T itself. Fig. 29(a) shows a template, T1 = { (a,0) (b,1) (c,2) (d,3) (e,5) }, containing 5 nodes. All CDFG edges have been omitted for clarity. (In the figures, we show behavioral template as a box containing one or more nodes in slots. The top slot in the box is offset 0, the second slot from top is offset 1, and so on. For example, the node "e" in Fig. 29(a) has

offset 5 in T1 because it is in the 6th slot from the top of the box.)

Whenever a node is a member of two or more different templates, we can always merge these templates into one. Consider the templates T1 and T2 in Fig. 29(a) and 29(b). If node g is to be added to template T1 at offset 1, then we merge T1 and T2 into T3 of Fig. 29(c).

4.0 Scheduling with Behavioral Templates

Instead of scheduling individual CDFG nodes, we restate the scheduling problem in terms of behavioral templates. Initially, we create one template for every CDFG node, and then merge the templates whenever nodes are added to other templates. This ensures that every CDFG node is a member of one and only one template. The timing constrained scheduling task is then to schedule all templates to minimize resource costs subject to timing constraints between templates. This section describes how we extend existing scheduling algorithms for behavioral templates.

4.1 Timing Constraints between Templates

From the CDFG, we construct a weighted, directed graph G=(V ,E) where V is the set of all behavioral templates in the CDFG, and E is the set of directed edges between templates. The weight $d(T_x, T_y)$ of an edge $e(T_x, T_y)$ in E specifies the minimum delay between the schedules of T_x and T_y, i.e.,

$$\text{schedule}(T_x) + d(T_x, T_y) <= \text{schedule}(T_y) \qquad \text{Eq.1}$$

The edges in E are constructed from the data/control dependencies between member nodes in the templates. For every pair of templates, T_x and T_y, $d(T_x, T_y)$ is the maximum value of

$w(n_i, n_j) + o_i - o_j$ Eq. 2

over all (n_i, o_i) in T_x and all (n_j, o_j) in T_y, where $w(n_i, n_j)$ is the minimum cycle delay from node n_i to node n_j.

Note that Eq.2 can be negative. This means $d(T_x, T_y)$ can be negative, and the graph G is not acyclic. If G contains any cycle of positive lengths, then the timing constraints are unsatisfiable. To check for positive cycles, we solve for the all-pairs-longest-path problem for G using a simple $O(N^3)$ algorithm, where N is the number of templates in G. The longest path lengths are stored in a matrix, LP, for subsequent incremental update of the as soon as possible (ASAP) and as late as possible (ALAP) schedules.

4.2 ASAP and ALAP Schedules

At the start of scheduling, we calculate the ASAP and ALAP schedules for all templates in G, to establish the scheduling time frame for each template. Since G may contain negative weighted edges, we use a relaxation algorithm similar to that in [3] to compute the initial ASAP/ALAP schedules:

(1) Propagate along positive edges in E only;

- for ASAP, propagate forward from the source of CDFG;
- for ALAP, propagate backward from the sink of CDFG;

(2) Relax schedules to satisfy constraints implied by negative edges in E;

(3) Repeat step 1 until no more changes in relaxation step.

When there are no positive cycles in G, the above algorithm is guaranteed to converge in e+1 iterations where e is the number of negative edges in E. The overall computational complexity is $O(N^2e)$ where N is the number of templates in G.

The ASAP and ALAP schedules define the initial time frames. Subsequently, as each template is scheduled, we update the time frames of other templates using the longest path lengths matrix, LP:

$$\text{schedule}(T_x) + LP(T_x, T_y) <= \text{schedule}(T_y) \text{ for all } T_x, T_y \text{ in } V$$

There is no need for relaxation in this incremental update because LP already takes into account all negative edges in E.

4.3 Cost Functions

We use a number of iterative/constructive scheduling algorithms each of which successively picks an unscheduled template and schedules it to a cstep in its time frame. The algorithms differ in how they pick the next template to schedule, and in how they pick which cstep to schedule the template to. We define the template priority/cost functions in terms of priority/cost functions on the CDFG nodes.

For example, in our implementation of list scheduling, the template priority function is defined as the maximum of its member nodes' priority values. This gives priority to the template containing the highest priority nodes. In our implementation of greedy scheduling, the incremental cost function for scheduling a template $T=\{ (n_i, o_i) \}$ to a cstep j, is defined as the sum total of the incremental costs for scheduling nodes n_i to csteps $j+o_i$.

Scheduling/de-scheduling moves on templates are implemented as moves on their member nodes. All data structures are updated as CDFG nodes are scheduled/de-scheduled. In particular, resource costs for functional units, registers and interconnects

are still computed according to the lifetimes and mutually exclusivity of CDFG nodes and edges. This approach is easy to implement and leverages previous work on scheduling CDFG nodes.

4.4 Pre-assigned Operations

Allowing negative edges in G requires that we extend scheduling algorithms to handle maximum timing constraints. This is complicated by "pre-assigned" operations, i.e., operations that are assigned to specific resources before scheduling. Examples of pre-assigned operations are memory read/write operations for the same RAM. We use a list scheduling algorithm to find an initial legal schedule based on source code ordering. However, list scheduling can fail to find a legal schedule when there are maximum timing constraints. So we augment list scheduling with a recovery step. When list scheduling fails, the recovery step relaxes the template schedules that caused scheduling failures, and iterates:

1. List scheduling step:

Successively consider operations in the ready list in increasing source code ordering. For each ready operation, n_i, check its template, $T_x = \{...(n_i, o_i)...\}$, for scheduling in the cstep s - o_i, where s is the current cstep. Postpone scheduling of T_x if any of the following is true:

- T_x has a "relaxed cstep" (see step 2) which is greater than s - o_i
- T_x has no "relaxed cstep", but ASAP is greater than s - o_i
- there is a resource contention if T_x is scheduled to s - o_i

When all nodes have been scheduled, exit with success.

If T_x is postponed due to resource contention, and if s - o_i is greater than or equal to the

ALAP cstep for T_x, then list scheduling has failed. When this happens, go to step 2 and try to recover.

2. Recovery step:

When list scheduling fails to find a legal schedule for T_x, we try to recover by increasing its ALAP cstep and rerun list scheduling in step 1. In order to increase the ALAP cstep for T_x, we find all scheduled templates, T_y, for which

$$alap(T_x) == schedule(T_y) - LP(T_x, T_y) \qquad \text{Eq. 3}$$

where $alap(T_x)$ is the ALAP cstep for T_x. For every such template, T_y, we set its "relaxed cstep" to $schedule(T_y) + 1$. This forces the next run of list scheduling to schedule T_y one cstep later.

This step exits with failure if any of the following is true:

- $ALAP(T_x)$ is at maximum global cstep
- there is no template T_y which satisfies Eq. 3
- algorithm has iterated for N times (N is the # of templates)

Figure 30 shows an example of this algorithm at work. In this example, CDFG nodes "a" and "c" are pre-assigned to the same resource. The source code ordering has "a" before "c" before "f". Initially, list scheduling in step 1 will schedule T1 to cstep 0, and then fails to schedule T2 because of resource contention at cstep 0, and because its ALAP cstep is 0 once T1 is scheduled to 0. In step 2, T1 will be assigned a relaxed cstep of 1. In the next iteration, list scheduling first schedules T2 to cstep 0, then schedules T1 to cstep 1 to avoid resource contention, and finally schedules T3 to cstep 3.

We have two recourses when the above algorithm fails: First, we can continue to try other scheduling algorithms which may still find legal schedules. Second, we can insert precedence constraints to sequentialize pre-assigned operations by their source code ordering. Any unsatisfiable maximum timing constraints would then be detected as positive cycles in the graph G.

5.0 Applications for Behavioral Templates

This section highlights how we use behavioral templates to advantage. Fig. 31 shows the overall flow of our scheduling process.

5.1 Inserting Timing Constraints

After extracting the CDFG and creating the initial templates, user-specified timing constraints are added to the CDFG. Minimum timing constraints are represented by precedence edges between nodes, but fixed timing constraints and maximum timing constraints are represented with the help of behavioral templates. Fixed timing constraints are when two or more operations must be scheduled in a fixed number of cycles apart. This is represented by adding one operation to the template of the other operation with the proper offset. For example, if n_j must start k csteps after n_i starts, and if n_i is in template $T= \{...(n_i, o_i)...\}$, then we add n_j to T at offset $o_i + k$ (Fig. 32(a)); if n_j must start k csteps after n_i *ends*, and n_i has a delay of d cycles, then we add n_j to T at offset $o_i + d - 1 + k$ (Fig. 32(b)).

However, if n_j must start k csteps after n_i ends, and n_i does not have a static delay (e.g., n_i is a subgraph), then we decompose the fixed constraint into a k-cycle minimum timing constraint from the end of n_i to the start of n_j, plus a k-cycle maximum timing constraint from the start of n_j to the end of n_i. This is shown in Fig. 32(c).

Note that in Fig. 32(c), we create a dummy *place holder* node, ph, and lock it in a

template with n_j (k cycles apart). This template combines with the precedence edge of weight 0 from ph to the end of n_j to represent the maximum timing constraint from the start of n_j to the end of n_j. The precedence edge of weight k from the end of n_j to the start of n_j, represents the minimum timing constraint from the end of n_j to the start of n_j.

In general, a maximum timing constraint of k cycles from a set of nodes, A, to the set of nodes, B, is represented by creating two place holders, ph1 and ph2, fixed k cycles apart in the template, and inserting a 0-weight precedence edge from ph1 to all nodes in A, and inserting a 0-weight precedence edge from all nodes in B to ph2. Fig. 37(a) contains an example of this where the dummy place holder nodes, t2 and t3, are used to lock a write operation to 0 cycle after the end of the loop.

5.2 Modeling Multi-cycle Operations

Behavioral templates also help model complex multi-cycle operations. When a single CDFG node is used to model a multi-cycle operation, it imposes some limitations due to CDFG semantics:

- Execution cannot start until ALL inputs are available.
- ALL inputs must be held stable throughout operation execution.
- ALL outputs are produced in the last cycle of execution,

This makes it difficult to model, for example, a 3-cycle RAM write operation where the address must be stable for the first two cycles, the data must be stable in the second cycle, and the write sequence finishes in the third cycle. To model such complex operations, we differentiate between combinational and sequential multi-cycle operations. A combinational operation has cycle synchronous inputs and outputs, so it is modeled by a single CDFG node. A sequential operation can have different cycle-by-cycle input/output connections and even resource requirements, and is modeled by several CDFG nodes that are locked into consecutive csteps by a behavioral template.

Fig. 33 shows the single-node and multiple-nodes-in-a-template models for the above 3-cycle RAM write operation. Note that in our model (Fig. 33(b)), the address and data inputs are de-coupled in terms of when and for how long each input must be stable. This also de-couples multiple outputs (if any).

This multiple-nodes-in-a-template model is even more powerful when resource requirements are added. For example, a pipelined operation uses different pipe-stage in different cycles, allowing overlapping pipelined operations to share the same hardware module as long as they do not have resource contention on any pipe-stage. If we view pipe-stages as internal resources and assign each pipe-stage a named "token", then we may label each node in the template model with the resource tokens it requires. As a node is scheduled for a cstep, we reserve its resource tokens for that cstep. The number of conflicting tokens (i.e., number of non-mutually-exclusive nodes that require the same token) in any cstep gives the number of pipelined modules needed in that cstep. Overlapping of pipelined operations can be scheduled on the same module because successive nodes in the template model require different tokens.

This removes assumptions about pipelined operations from the scheduling algorithms. We may now model operations on complicated pipelines, and template-based scheduling will properly schedule these operations on the pipelined modules. Fig. 34 shows examples of operations on pipelines with internal feedback, sequential inputs, and multiple outputs. We use "a[s1]" to denote an operation named "a" which requires the token "s1".

Actually, resource tokens need not correspond to physical hardware resources, but may be considered a more general mechanism for specifying how different types of operations can overlap in time on the same module. Consider a 2-cycle RAM which has one read-port and one write-port, whose read/write cycles must be synchronized. Fig. 35 shows how we use resource tokens to specify this constraint to scheduling. If two such operations are pre-assigned to the same RAM, then resource contention of

any of "s1", "s2", "s3" and "s4" implies an illegal schedule. The token "s3" prevents read operations for the same RAM to overlap; the token "s4" prevents write operations fro the same RAM to overlap; and the tokens "s1" and "s2" prevent read and write operations for the same RAM from being scheduled exactly one cycle apart.

To handle multi-port RAM's, we allow a module to carry more than 1 copy of a given resource token. For example, to model a 4-port RAM where each port can be used fro both read or write, we would define the RAM module to have 4 "r/w" tokens, and model read and write operation on this RAM to require 1 "r/w" token each. This would allow scheduling to perform up to 4 simultaneous read or write operations on the same module.

5.3 Pre-Chaining

Just before scheduling, we selectively force operation chaining by locking operations in the same cycle using behavioral templates. This "pre-chaining" step reduces scheduling complexity at the expense of scheduling freedom. User-specified chaining directives are applied in this step. We also implement automatic pre-chaining for logic operations and bit-manipulation operations to save registers.

Logic operations include bit-wise AND, OR, NOT, EXOR operations, and bit-manipulation operations include bit-extract, bit-concatenate, constant bit/word generator operations. These operations are good candidates for pre-chaining because they have small propagation delays and they are not resource shared. Thus pre-chaining can be done on the basis of register costs alone. We implement a greedy algorithm for pre-chaining:

1. In a forward traversal of the data flow graph, pre-chain a logic/bit-manipulation operations with its predecessors if there are fewer output bits than input bits;

2. In a reverse traversal of the data flow graph, pre-chain a logic/bit-manipulation operation with its successors if there are fewer input bits than output bits.

3. Iterate until there are no more changes.

Fig. 36 shows examples of good pre-chaining configurations.

5.4 Hierarchical Scheduling

As shown in Fig. 31, our extracted CDFG is hierarchical, in which each level of the hierarchy corresponds to a loop or a subroutine. Hierarchical scheduling proceeds in a bottom-up traversal of the hierarchy. At each level, instead of representing subgraphs as super nodes, we inline each subgraph and use a behavioral template to interlock the inlined nodes according to the subgraph's schedule.

Inlining subgraphs allows certain boundary optimizations. First, unused subgraph outputs (and the operations that produce these outputs) can be deleted. This deletion can recurs to unused subgraph inputs and then to operations that fed these inputs. Second, inlining subgraphs allows scheduling of neighboring nodes to take advantage of when individual subgraph inputs/outputs are actually required/produced, whereas representing subgraphs as super nodes would force scheduling to assume that all subgraph inputs/outputs are required/produced in the same cycles.

Another advantage of inlining subgraphs is that scheduling maintains accurate cycle-by-cycle resource costs. This allows, for example, to calculate resource costs of scheduling operations in the first and last cycles of a loop subgraph. (When an outside operations is scheduled in the first/last cycle of a loop it is performed when entering/exiting the loop.

In fact, hierarchical scheduling is used to implement sequential multi-cycle operations.

In the initial CDFG, each sequential operation is a subroutine call to some library function, which is a pre-scheduled CDFG whose nodes are labeled with the required resource tokens. During inlining, each sequential operation is replaced by an inlined copy of its function's CDFG, and a template is created to lock these inlined nodes. This creates the multiple-nodes-in-a-template model for sequential operations.

The disadvantage of inlining subgraphs is that more nodes are scheduled instead of a small number of super nodes. This is balanced somewhat by the fact that inlined nodes are grouped by templates into a few scheduling units, so at least the scheduling solution space is not much bigger.

6.0 Results

Behavioral templates have been implemented in the Synopsis Behavioral Compiler™ product. Behavioral Compiler™ inputs a VHDL or Verilog behavioral description, performs scheduling, allocation, module selection, binding, and control optimization, and outputs a RTL design which is then optimized by RTL optimization [2], FSM optimization, and logic synthesis.

We will use "dft", a discrete fourier transform design, to illustrate behavioral templates. On reset, dft sequentially reads in the real and imaginary parts of the coefficients into arrays cmem and dmem. These arrays are mapped to the memory "CRAM" (cmem in the lower bank, dmem in the upper bank). It then enters the main processing loop. In each iteration, dft signals it is ready for processing, do a busy wait for the "start" signal, and then sequentially reads in the real and imaginary parts of the data points into arrays amem and bmem, which are also mapped to two halves of a memory, "DRAM". It then enters two nested FOR loops which compute the discrete fourier transform values and write them out. The memories are two-cycle RAM's whose read/write models are shown in Fig. 35. The multiply operations are done on a 2-stage pipelined multiply.

Fig. 37 shows the CDFG fragment for the busy-wait on the "start" signal. Fig. 37(a) shows the initial CDFG containing the fixed constraints that the "ready" signal be asserted one cycle before the busy wait, and deasserted 0 cycle after the busy wait. Fig. 37(b) shows the same CDFG fragment that is finally scheduled. By this time, hierarchical scheduling has already scheduled the busy wait loop, and the loop body is inlined and locked in a template. Also, pre-chaining has locked the constants with the write operations.

However, the main scheduling problem is in the inner-most loop, which reads from memories the complex data (*a*, j*b*), and coefficient (*c*, j*d*), and computes *psum* += (*a*c – b*d*) and *ipsum* += (*a*d + b*c*). Table 1 shows the scheduling and allocation results for the computation part of this loop. Note the pipelined RAM read's are chained with the pipelined multiply operations.

amem bmem	cmem dmem	p-mult	+/-
r0[s1]	r6[s1]		
r1[s2] r2[s1]	r4[s1] r7[s2]	x0[s1]	
r3[s2]	r5[s2]	x1[s2] x2[s1]	
		x4[s1] x3[s2]	add1
		x5[s2] x6[s1]	add2
		x7[s2]	sub1
			add3

TABLE 1. Scheduling/allocation results for dft's inner loop

FIGURE 46. Scheduling/allocation result for computation part of dft's inner-most loop

In Table 2 and 3, we present some design statistics. #line is number of VHDL/verilog lines in the source. #loop is number of loops with nesting levels in brackets. #node is number of CDFG nodes. #template is total number of templates scheduled. The ratio of nodes to templates are shown in brackets. #RAM is the number of on-chip memories used. #gate is gate count after logic synthesis (excluding RAM's). Note the difference

between #node and #templates.

Table 2 presents several HLSW benchmarks, modified to use more realistic bit-widths. EWF is the fifth order elliptic wave filter example (20 csteps for the main loop, using 1 16X16 pipelined multiply, 2 32-bit adders, 10 32-bit registers and 1 16-bit register). KF is the Kalman filter modified to use 5 RAM's.

	#line	#loop	#node	#template	#RAM	#gate
EWF	128	2 (2)	103	78 (1.32)	0	9677
KF	207	10 (4)	261	144 (1.81)	5	7963
i8251	596	54 (3)	1238	625 (1.98)	0	5274
gcd	57	2 (2)	77	22 (3.50)	0	825

TABLE 2. Design statistics for several HLSW benchmarks

	#line	#loop	#node	#template	#RAM	#gate
dft	218	5 (3)	151	65 (2.32)	2	3920
rgb filter	247	3 (2)	286	109 (2.62)	4	4316
idct	274	8 (3)	902	287 (3.14)	3	32677
viterbi	262	3 (2)	1797	296 (6.07)	0	3653
graphics ctrl	123	2 (2)	98	30 (3.27)	0	4071
high pass	389	5 (2)	310	153 (2.03)	11	8970

TABLE 3. Design statistics for several industrial examples

Table 3 lists several industrial examples. Compared to benchmark examples, these designs tend to:

- have more complicated reset sequences before the main processing loop,
- have more cycle-by-cycle timing constraints on IO operations,
- have more logic operations and bit-manipulation operations,
- use multiple RAM's or multi-port RAM's to improve RAM access bottlenecks,
- use pipelined operations to increase throughput.

7.0 Conclusion

In this paper, we have presented our work on scheduling using behavioral templates. The most important value of behavioral templates is that they enable simple solutions to the problems of (1) enforcing fixed and maximum timing constraints, (2) modeling complex sequential operations, (3) pre-chaining of logic and bit-manipulation operations, and (4) hierarchical scheduling. For this reason, behavioral templates have been instrumental in our productization of behavioral synthesis.

Future work will investigate adding structural templates to partition the design based on the structural regularity.

8.0 Acknowledgement

We would like to acknowledge Russ Segal and Dennis Fogg, who designed and implemented library interface for our sequential operations. We would also like to acknowledge Pradeep Fernandes and Hazem Almusa for helping us collect the results reported in Tables 1, 2, and 3.

9.0 References

[1] R. Camposano, "Path-based scheduling for synthesis", IEEE transactions on Computer-Aided Design, vol. CAD-10, pp85-93, Jan 1991.

[2] B. Gregory, D.MacMillan & D. Fogg, "ISIS: A System for Performance Driven Resource Sharing", in Proc. of 29th DAC, pp285-290, June 1992.

[3] D. Ku & G. De Micheli, "Relative Scheduling under Timing Constraints", in Proc. of 27th DAC, pp59-64, June 1990.

[4] M.C. McFarland, A.C. Parker & R. Camposano, "The High-Level Synthesis of Digital Systems", in Proc. of IEEE, vol. 78, pp301-318, Feb. 1990.

[5] P. Michel, U. Lauther and P. Duzy, "The Synthesis Approach to Digital System Design", Kluwer Academic Publishers, 1992.

[6] S. Note, W. Geurts, F. Catthoor & H. De Man, "Cathedral III: Architecture-Driven High-Level Synthesis for High Throughput DSP Applications", in Proc. 28th DAC, pp597-602, June, 1991.

[7] M. Nourani & C. Papachristou, "Move Frame Scheduling and Mixed Scheduling-Allocation for the Automated Synthesis of Digital Systems", in Proc. 29th DAC, pp99-105, June, 1992.

[8] D.S. Rao & F.J. Kurdahi, "Partitioning by Regularity Extraction", inProc. 29th DAC, pp235-238, June, 1992.

[9] D.S. Rao & F.J. Kurdahi, "System Modeling for High-Level Synthesis Using Regularity Extraction", D.S. Rao & F.J. Kurdahi, in Proc. Sixth International Workshop on High Level Synthesis, pp267-272, Nov. 1992.

[10] K. Rose & C.T. Chang, "Cluster-Oriented Scheduling in Pipe-lined Data Path Synthesis", in Proc. ICCD, Oct. 1993.

App ndix B

Behavioral Synthesis Methodology for HDL-Based Specification and Validation

D. Knapp, T. Ly, D. MacMillen, R. Miller

Synopsis Inc.
700B E. Middlefield Road
Mountain View, CA USA 94043

Abstract

This paper describes a HDL synthesis based design methodology that supports user adoption of behavioral-level synthesis into normal design practices. The use of these techniques increases understanding of the HDL descriptions before synthesis, and makes the comparison of pre- and post-synthesis design behavior through simulation much more direct. This increases user confidence that the specification does what the user wants, i.e. that the synthesized design matches the specification in the ways that are important to the user. At the same time, the methodology gives the user a powerful set of tools to specify complex interface timing, while preserving a user's ability to delegate decision-making authority to software in those cases where the user does not wish to restrict the options available to the synthesis algorithms.

1.0 Overview

This paper describes a synthesis methodology that uses high-level synthesis (HLS) of behavioral hardware-description language (HDL) descriptions. HLS has the distinguishing characteristic that operations are automatically *scheduled*, i.e. assigned to states, as opposed to lower-level synthesis, in which operations are assigned to states by the user [1, 2, 3]. For example, in an HDL description of a square root function, an operand *x* would be loaded, a series of operations would follow, and a

single result *r* would be returned. The read *x* and the write *r* might be fixed to particular states or times by a communication protocol, but the internal operations that compute the square root would be automatically scheduled.

A prospective user of HLS will then ask a number of questions. These will likely include the following:

- How can I constrain I/O operations to fall into particular cycles, or range of cycles, to meet existing protocols?
- How can I constrain I/O operations to have particular timing relationships? For example, how can I constrain a data ready strobe to be synchronous with data on data ports?
- How can I be confident that my interface timing specification really works with the surrounding hardware?
- How can I give the scheduling software optimization opportunities when my timing specification is not rigid? For example, I might not care exactly when data was transferred, as long as a corresponding strobe remains synchronized with the data. Thus the strobe and data should be locked together, but the locked strobe/data pair of operations could move.
- How can I be confident that the synthesized hardware will really do what I want it to:

    1. In the sense that it computes the right result,

    2. In the sense that scheduling of I/O operations does not "break" its I/O protocols.

These questions can be reformulated as requirements on the HDL description methodology to be used in conjunction with HLS:

- The original HDL description should be simulatable.
- There should be a mode wherein the cycle by cycle I/O timing of the original HDL description is preserved exactly; i.e., no I/O timing difference will be allowed between the pre- and post-synthesis designs; it will also allow the user to meet the most rigid cycle-based timing protocols.
- There should be a mode wherein timing relationships between I/O signals can be simply and easily preserved across synthesis, but where 'stretching' (cycle level delay insertion) is permitted, so that the user does not have to specify exactly how many cycles a computation will take. This mode should allow manual constraints. Such a mode allows comparisons of pre- and post-synthesis I/O timing between "similar points" of the pre- and post-synthesis waveforms.
- There should be a mode in which the user explicitly specifies all timing constraints without reference to the simulation behavior of the HDL; the only timing constraints inferred from the HDL description are ordering constraints among I/O operations sharing a port. This mode gives the greatest flexibility, both for optimization and for specification of complex timing relationships; it is also the most difficult to use.

We call these three modes the *cycle-fixed* IO *scheduling mode*, the *superstate-fixed IO scheduling mode*, and the *free-floating IO scheduling mode* respectively. Each has consequences for the style of HDL description and validation methodology. These modes give the user a wide range of choices in specifying I/O timing, with a corresponding range of ways in which validation of the specification and comparison of the implementation with the specification can be performed.

1.1 Structure of this paper

The balance of this paper is structured as follows. In Section 1.2, related work in this field is discussed. Following that, in Section 2, some mode-independent considerations

and assumptions are described. In Section 3, the cycle-fixed mode is described in detail. Then in Section 4, the superstate-fixed mode is described. In Section 5, the free-floating mode is described. In Section 6, experience with the current software is described; finally, in Section 7 the paper is summarized and conclusions are drawn.

1.2 Related Work

High-level synthesis has been well described in the literature; see, for example, Camposano[1], Gajski[2], Maerz[3]. These tutorial papers describe the basics of HFL systems. CALLAS [4] describes work in the area of maintaining simulated behavior that is exactly the same pre- and post-synthesis; this idea is reflected in the cycle-fixed mode described here. The superstate-fixed mode is related the High Level State machine of [5], and to the behavioral finite state machines (BFSM's) of [6]. Our approach of validation through simulation is typical of current industry practice; it complements, but cannot completely replace, more formal methods [7].

2.0 Basic assumptions

The circuit to be synthesized by HLS consists of a collection of always blocks (VHDL processes); each always block will be mapped to hardware consisting of a datapath and a control FSM. Each will be synthesized separately.

Control over timing makes use of clocking statements in the source HDL. In Verilog, this can be done by use of @(posedge clock) or @(negedge lock) statements (in VHDL "Wait until clock' event and clock = '1';" gives us a rising-edge clock). These are used to separate I/O events that are to happen in different clock cycles. Event triggers using other signals are specifically disallowed, with the exception of asynchronous reset and a special gating methodology described in Section 2.2, used for synchronizing I/O.

2.1 Reset

In order to handle resets in an intuitively appealing way, we call attention to the always block (VHDL process) that will be scheduled. In our methodology this block contains a single all-encompassing, nonterminating loop, here called *reset_loop.*

```
always begin: b1
    begin: reset_loop
        // reset sequence behaviors
        forever begin
            // normal mode behaviors
        end
    end
end
```

Inside *reset_loop* is a *reset sequence*; this consists of all behaviors associated with reset. For example, in a microprocessor the reset sequence would clear the program counter, disable interrupts, and initialize the stack pointer. The reset sequence may contain many clock cycles, e.g. to initialize a RAM. Following the reset behavior is the 'normal mode' loop, which does not terminate either; this loop contains behaviors that are executed until the next reset occurs. In a microprocessor, for example, the normal mode loop would be the fetch / execute cycle.

In order to simulate the effect of synchronous resets correctly in the source HDL description, the user must insert a statement of the form (in VHDL this would be "when reset = '1' exit reset_loop")

```
if(reset == 1' b1) disable reset_loop;
```

after every @posedge statement. This disable has the effect of restarting the block (process) following a clock edge upon which reset is found to be true. Simulation of

synchronous resets can be matched both pre- and post-synthesis.

Another capability can also be provided in which the user declares a reset pin to the synthesis software, which then synthesizes the reset; but because the reset behavior is not encoded in the HDL, resets cannot be simulated correctly before synthesis using this technique. Scheduling cannot handle exits triggered by a reset in the same way as other exits, because there may be read-before-write accesses in the HDL. Consider the following: In this situation, the assignments

```
begin: reset_loop
    outport <= x; // x is read before write!
    begin: main-loop
        x = v1;
        @(posedge clock);
        if (reset == 1'b1) disable reset_loop;
        x = v2;
    end
end
```

of x cannot be rescheduled, because this would change the observable behavior of the circuit immediately following a reset pulse. If, for example, the second write to *x* was rescheduled before the clock edge, then the output immediately following a reset pulse would be *v2* in the scheduled design; but it would be *v1* in the original description. So if we are to allow read before write in the HDL, we must either relax the requirement that all behaviors must be identical, or we must forbid movement of such side effects across clock boundaries. Side effects on variables that are always written before they are read are not affected.

2.2 Registered outputs

VHDL signals and Verilog reg variables behave like register or latch outputs. That is, they hold their values once set. For implementation reasons, we chose to register all outputs of HLS synthesized designs; thus a nonblocking (signal) assignment becomes a register write. This has the consequence that responses to external events cannot happen until the cycle after the external event, as show in Fig. 38.

Figure 38 shows the behavior of a synthesized circuit where the HDL input is of the general form

```
if (Ready == 1'b1) then Data <= foo;
@(posedge clock);
```

This timing corresponds to both input and output. Notice that this timing diagram implies that the control FSM for the synthesized data path is a Mealy machine; and that the overall synthesized design is a Moore machine.

Here is an example combining an asynchronous reset and a compact busy wait on a data strobe.

```
while (strobe != 1) begin
    @(posedge clock or posedge reset);
    if (reset == 1'b1) disable reset_loop;
end
```

3.0 Cycle-fixed mode

High-level synthesis in *cycle-fixed* mode can be described by the following statement:

- Cycle-by cycle I/O timing is identical between the pre-and post-synthesis designs

This means that validation by simulation is straightforward: a user need merely simulate

the pre-and post-synthesis designs side by side, and check for differences in the outputs. Alternatively, the synthesized design can be inserted into the original test bench without modifying the test bench. The only differences that are visible involve combinational delays in the form of setup and hold times; for example, a delta-delay setup time would become a real setup time, and a registered output pin will not transition exactly on the clock edge, as it would in the pre-synthesis simulation (in zero-delay simulation one should ensure that data transitions occur slightly after clock transitions; failing to do this is the most common source of simulation mismatches. The problem comes about because of varying numbers of simulation-cycle delays in the clock and data wires of the circuit: the clock can arrive 'after' the data by an infinitesimal (zero-time) amount. This causes something analogous to a setup-time violation). This is shown in Fig. 39.

Notice that this mode only constrains the I/O operations of the design. That is, the reads and nonblocking (signal) writes of the HDL are tied to particular cycles. But his still leaves optimization opportunities for the scheduling algorithm: other operations (e.g. additions, memory operations, and register read reads and writes) can be shifted in time, as long as they consume data after it has been read in, and produce data in time to write it out. The I/O operations provide a series of 'stakes in the ground' that define time frames within which all other operations are free to move.

The main advantage of cycle-fixed mode is that the user can synthesize exactly the same timing diagram that the original HDL specification shows in simulation; thus, if the simulated HDL specification works in a particular context, then the synthesized design will also work, assuming only that setup, hold, and propagation delays, etc. as shown in Fig. 38b meet the clock cycle time.

A further advantage of cycle-fixed mode is that simulation of a zero-gate-delay model of the synthesized design will match the original specification exactly; hence a simple file difference program can be used to compare pre- and post-synthesis designs. This is

expected to have a profound effect on user acceptance of HLS as a viable tool in the design cycle: users are able to simply and efficiently check the equivalence of designs before and after synthesis.

There are a number of methodological and implementation considerations that affect the way we can write and implement cycle-fixed mode. These will now be described.

3.1 Numbers of clock edges

One consequence of the commitment to maintain exact I/O equivalence in cycle-fixed mode is that numbers of clock edges cannot be varied inside the scope of loops and conditionals. To do so would distort the I/O timing of the design.

3.2 Loop boundaries

Every loop of an always block must contain at least one clock edge statement. The only exception to this is loops with constant iteration bounds, which can be unrolled during synthesis.

A loop can be thought of as a subgraph of a finite-state machine (FSM) which forms a cycle. The synthesized design will enter this cycle when the loop is executed, and leave it when the loop is exited. Such a loop is shown in Fig. 40.

The loop of Fig. 40 corresponds to the state labeled 'Loop'. During each pass of the loop, the value of v2 will be written to the outport *o*.

The main consequence of matching this behavior is the splitting of the conditional test *c*. Notice that it was necessary, in order to capture the timing of the original, to have a state transition that bypassed the loop altogether if *c* was false when it was first tested. This means that the test must be performed in two places: once in state *prev*, and once

in state *loop*. In general, it is necessary to unroll the first state of the first pass through a while loop in order to capture this behavior correctly.

If we wish to avoid unrolling the first pass, then it is necessary to rewrite the loop so that (1) there is a clock edge on all paths between the writes of *o1* and *o3*, and (2) there is a clock edge between the conditional test and any succeeding I/O, as shown in Fig. 41.

3.3 Conditional multicycle operations

A *multicycle* operation is one that has a longer combinational delay than the clock cycle. This imposes special constraints on synthesis in cycle-fixed mode, because it is necessary to stabilize all data and control inputs to the hardware block that implements the multicycle operation. This includes all the control inputs of all multiplexers that drive multicycle operations; clearly we cannot afford glitches on these paths.

But inserting these registers means that we need to know what to strobe into the registers one cycle before the multicycle operation is to begin. Thus we need to add extra time, under some circumstances, so that the stabilizing registers can be properly loaded. This is illustrated in Fig. 42.

Notice that we needed three clock cycles to do this properly; one to get the condition and strobe the stabilizing registers, and two to perform the multicycle addition. Notice also that such delays can often be hidden, where the multicycle operations are not constrained by I/O; but that in this case there is no opportunity to hide the additional delay associated with stabilizing the inputs.

3.4 Loop pipelining in cycle-fixed mode

Loop pipelining is a technique whereby a loop can be made to act like a pipeline. Thus the loop has a relatively long latency, i.e. the time from a data input to the

corresponding data output; and a shorter initiation interval, which is the rate at which data can be delivered to and read out from the loop. In cycle-fixed mode, and with some extra constraints in the other modes, a simple way to imply loop pipelining while maintaining timing equivalence is to use a delayed assignment (in VHDL, a transport delay) on the output statement. Suppose, for example, we have a loop whose latency is ten cycles, but whose initiation interval is two cycles; we can put an output write after the second clock edge statement, with a delay of eight cycles. This will simulate the same way both before and after synthesis.

```
    while (condition) begin
        @(posedge clock); // 10 ns clock
        @(posedge clock);
        out <= #80 value; // delayed by 8 cycles
    end
```

4.0 Superstate-fixed Mode

The *superstate-fixed* I/O mode is used where the I/O should inherit its general structure from the HDL, but where there is some freedom to shift I/O operations in time. Consider, for example, the two-wire handshaking protocol shown in Fig. 43. The two-wire protocol is insensitive to the time between transitions; this makes it ideal for many applications. In a case like this, the only things we really need to assure in order to have correct timing are that (1) the signal transitions occur in the right order, and (2) that the transitions of *Strobe* and *Data* maintain a lockstep relationship. Beyond that, the user might not care very much how many clock cycles were inserted by scheduling; other design optimization criteria (such as the number of gates to compute the data value) might dictate more or fewer clock cycles for this transaction. The cycle-fixed mode is unsuitable for this kind of loosened specification of timing: the user could be forced to edit the code many times, with varying numbers of clock edge statements each time, looking for the best implementation.

The *superstate-fixed I/O scheduling mode* can be expressed by the following statements:

- Adjacent pairs of clock edge statements in the HDL form the boundaries of superstates.
- All I/O operations in a superstate remain in that superstate.
- A superstate may be expanded by the scheduler, which can add clock cycles to lengthen a superstate.
- All I/O writes in a superstate will always take place in the last clock cycle of the superstate.
- I/O reads may float within a superstate.

These rules, taken together, mean that an HDL scheduled in superstate mode will show the same signal transitions and ordering as the original HDL; but that the original timing may potentially be 'stretched' by the addition of new clock edges. This is illustrated in Fig. 44, where the original HDL simulation of an I/O transfer taking three cycles has become five cycles long by the addition of two extra cycles to the second superstate.

4.1 Protocols in superstate mode

One of the major advantages of superstate mode is that handshaking I/O protocols are not distorted by the addition of clock cycles to superstates. This has two beneficial consequences: first, comparison of simulated pre- and post-synthesis designs is straightforward; and second, protocols that are insensitive to increased numbers of clock cycles will not be 'broken' by superstate scheduling. Hence if a design consists of many processes, each of which is to be scheduled, the use of handshaking communication in conjunction with superstate mode scheduling will ensure that the design will continue to work after synthesis. The same considerations apply to the

simulation test bench as well: the test bench must communicate with the synthesized design(s) via handshaking protocols; otherwise it may have to be modified to communicate successfully with the synchronized design. This happens because the read and write operations occur at different times pre- and post-synthesis; the test bench must be able to tolerate this, or the user will have to retime the test bench.

Protocols that do not involve explicit requests and acknowledges can still be used; but care must be taken with data to be read in by the synthesized process. In particular, recall that read operations may move freely within their superstate. This means that data being presented to the synthesized circuit must be either valid during the entire superstate in which it is read, or else retimed after scheduling. This will ensure that the read operation will always get the correct data.

4.2 Constraints in superstate mode

The reason a designer would use superstate mode instead of cycle-fixed mode is that some part of the schedule does not have a fixed timing bound, and the user does not want to imply such a bound by using cycle-fixed I/O. However, the user may have a non-handshaking protocol, or a protocol that streams data once synchronization has been established by the protocol. In such cases the parts of the schedule that perform synchronization may need to be handled as if the scheduler was in cycle-fixed mode; while the other parts of design can be allowed more freedom. For example, consider the fragment

```
    while (ready == 1'b0) begin: handshaking_loop
        @(posedge clock);
    end
    @(posedge clock);
    a1 = in_port; // label read_1
    @(posedge clock);
```

```
    a2 = in_port; // label read_2
    @(posedge clock);
    out_port <= long_involved_function(a1, a2);
    out_ready <= 1'b1; // label done
    @(posedge clock);
```

Here the external logic provides the data for *read_1* and *read_2* in the two cycles after the signal *ready* goes true; the synthesized system must pick it up then, or the protocol will be broken. Furthermore, insertion of extra cycles in the loop *handshaking_loop* will cause the interface to behave unpredictably. Thus cycle-fixed mode would seem to be indicated. However, suppose that there is no need for the output to show up until 20 cycles after the input has been delivered; the designer will thus want to allow the scheduler authority to add cycles to the last superstate, and rely on a test of the *out_ready* pin to synchronize the data on *out-port*. Thus stretching can be allowed in the last superstate, but not in the first three.

This can be done by means of explicit point-by-point scheduling constraints; that is, constraints that tie two labeled operations together in a particular timing relationship. A constraint set that would serve the purpose is

1. The time from the beginning of *handshaking_loop* to its end should be exactly one cycle.
2. The time from the end of *handshaking_loop* to the beginning of *read_2* should be exactly one cycle.
3. The time from the end of *handshaking_loop* to the data ready strobe *done* is no greater than 21 cycles.

Notice that these constraints are not part of the HDL; but they are a necessary part of the methodology. They can be implemented as pseudo-comments, as attributes, or as directives in a scheduler command file. Notice also that they can be applied to non-I/O

operations as well, in all three modes, to give the user a little extra control over the scheduling process.

4.3 Superstate HDL methodology

Superstate mode defines superstates as containing the I/O operations that fall between adjacent pairs of clock edge statements. This definition has the consequence that sometimes an HDL prepared for superstate mode needs clock edge statements that are not needed in cycle-fixed mode. For example, the text of Fig. 40 is ambiguous when the HDL is considered as input for superstate mode. This comes about because two writes are separated by a conditional @posedge. If the loop condition is true, then the writes should be in different superstates; if it is false, then they should be in the same superstate. Clearly there is no unique static assignment of I/O operations to superstates in this situation.

Furthermore, there is an implicit ordering of operations conferred by the sequencing of the HDL text; this ordering cannot be allowed to come into conflict with the ordering conferred by the migration of reads into any cycle of their superstate and writes into the last cycle of their superstate.

The HDL methodology rules that prevent ambiguities and contradictions in superstate mode are:

1. A superstate that contains a loop continue is called a continuing superstate. Implicitly, the last superstate of a loop is also a continuing superstate. A continuing superstate and the first superstate of the loop are really the same superstate; there is no clock statement on the execution path going from one to the other. If a continuing superstate contains a write, then the first state of the loop cannot contain any I/O, because a write belonging to the continuing

superstate would be migrated to the end of the first loop superstate: this would result in a violation of the HDL's ordering constraints.

2. A superstate that contains a loop beginning cannot include both an I/O write before the loop beginning and any I/O operation inside the loop. For example,

```
@(posedge clock);
out_port <= write1_data;
while (cond) begin
    read1_data = in_port; // Illegal!
    @(posedge clock);
    ...
end
```

the write in this fragment conflicts with the read in the beginning of the loop; they are the same superstate.

3. A write cannot precede a while loop that is succeeded by any I/O operation, unless there is a clock edge statement between either the write and the loop begin, or between the loop end and the second I/O operation.
4. A loop having a superstate in which both a loop exit (other than a reset exit. Reset exits can be ignored after a preprocessing step in which they are detected and global reset behavior is enacted, as explained in Section 2) and an I/O write are located must have a clock edge statement between the loop end and the next I/O operation.
5. A conditional clock edge (e.g. an @edge on one branch of a conditional) cannot be used to separate a write from another I/O operation. This fragment is illegal for that reason.

```
out_port <= v1;
if (cond) @(posedge clock);
v2 = in_port;
```

5.0 Free-floating I/O mode

It will sometimes be the case that a user will need to convey more freedom to the scheduler than is allowed by the superstate I/O mode. For example, the user may wish to allow two unrelated writes to be permuted. Consider the fragment of Fig. 45. In this situation, the user might not care whether the first or the second function happens first; indeed, they could be interleaved and the user might not care. But neither superstate nor cycle-fixed mode will permit permutation of I/O operations and waits; so a more powerful mode is needed.

The *free-floating* mode is characterized by implicit constraints on single I/O ports and explicit user constraints.

Implicit I/O port constraints are derived directly from the HDL text and are imposed on the sets of reads and writes that occur on a single port. These are formed into partially ordered sets, one for each port, where the ordering is derived from a static execution trace analysis for the source HDL. The schedule constructed by synthesis can only transpose two members of one of these sets if there is no ordering relationship between them.

This, however, says nothing about ordering of reads and writes that occur on different ports, which must be explicitly constrained by the user, by means of the explicit two-point constraints described in Section 4.2.

For example, in our experience a common early mistake in free-floating mode is to expect a data strobe's timing to be fixed with respect to that of the data being strobed. This will not necessarily be the case if the user does not issue explicit constraints.

The downside of this mode is the number of explicit constraints that the user must construct. This can easily be comparable in numbers of lines to the HDL input itself. In addition, it is very easy to get such constraints wrong, or to forget a crucial constraint; hence the cycle-fixed and superstate modes are simpler and less error-prone to use.

6.0 Experience

Support for the methodologies discussed above has been built into a commercial product, the Synopsis Behavioral Compiler™. This product is currently in use at a number of sites. Of these, about half use Verilog as their input HDL; the rest use VHDL.

Experience to date indicates that the superstate mode is usually the most convenient from the standpoint of ease of specification of complex timing behaviors. The next most convenient is usually the cycle-fixed mode. The reason for this is that the power of the free-floating mode comes at the price of manually added constraints; while the cycle-fixed mode requires the user to add clock cycles to the source HDL when, e.g., the duration of a particular loop is to be changed.

From the standpoint of ease of validation of results, the cycle-fixed mode is usually a little more convenient than the superstate mode. This is because the handshaking protocols necessary to get the design talking to the test bench after superstate-mode scheduling must be designed and written in both the test bench and the specification; or alternatively the test bench timing must be modified to match the schedule of I/O of the post-synthesis design.

One area in which the free-floating mode seems to be more convenient than the others is in that of exploration. Here the user is more interested in getting a rough idea of the cost and speed of a design or algorithm, than in getting its interfaces exactly right. In this context, the ease of turning the design around and the high degree of freedom from methodological constraints makes it simpler to change the design and resynthesize to see what the overall results are. Then when the general outlines of the algorithms, representations, etc. are clear the user can begin to worry about the detailed I/O timing.

The overall effort of getting I/O interfaces right using these three modes is usually less

than the effort spent in getting the best possible quality of results. Even with behavioral synthesis, HDL writing styles still can have a large impact on the quality of the synthesized circuit. Examples that can affect synthesis quality are: loop ordering, assignment of variables and arrays to memories, choice of loop pipeline initiation intervals and latencies, pipelined components, embedding combinational logic in reusable function blocks, the tradeoff between multicycle operations and fast clock rates, and the partitioning of the design into datapath/controller subunits (i.e. always blocks; in VHDL, processes). All are potentially of great importance to the quality of results, and all represent true engineering decisions that must be carefully considered if a really good design is to be achieved.

7.0 Conclusion

We have presented HDL methodologies for the synthesis of various kinds of I/O timing and protocols, and for simulation-based validation of the synthesized design against the original specification. Three modes of scheduling I/O operations have been presented:

1. Cycle-fixed, in which the design has exactly the same cycle-level I/O timing before and after synthesis;
2. Superstate-fixed, in which I/O operations are grouped by pairs of @posedge statements; post-synthesis timing behavior is a (potentially) stretched version of the pre-synthesis timing; and
3. Free-floating, in which the only constraints on I/O scheduling are either between operations sharing a port or supplied by the user.

Some of the implications of the scheduling modes were described. In the cycle-fixed and superstate modes, these involve the placement of clock edge statement, loop boundaries, conditionals, and I/O operations; while in the free-floating mode there are no rules of this kind.

Experience with production software which implements these methodologies has been described, and conclusions based on that experience have been drawn.

8.0 References

1. R. Camposano, W. Wolf. "Trends in High-Level Synthesis," Kluwer, 1991.
2. D. Gajski, N. Dutt, A. Wu, S. Lin. "High-Level Synthesis: Introduction to Chip and System Design." Kluwer 1992
3. S. Maerz, "High Level Synthesis." In *The Synthesis Approach to Digital System Design*, P. Michel, U. Lauther, P. Duzy, eds., Chapter 6. Kluwer, 1992.
4. A. Stoll and P. Duzy, "High-Level Synthesis from VHDL with Exact Timing Constraints," *Proceedings from the 29th ACM/IEEE Design Automation Conference*, pp. 188-193, IEEE, 1992.
5. R.A. Bergamaschi, A. Kuehlmann, S-M. Wu, V. Venkataraman, D. Reischauer, and D. Neumann, "A Methodology for Production Use of High level Synthesis." workshop Proceedings, Sixth International Workshop on High Level Synthesis, (1992).
6. W. Wolf, S. Takach, C.-Y. Huang, R. Manno, E. Wu. "The Princeton University Behavioral Synthesis System." *Proceedings of the 29th ACM/IEEE Design Automoation Conference*, pp. 182-187, IEEE, 1992.
7. K.L. McMillan, Fitting Formal Methods into the Design Cycle, *Proceedings of the 31st ACM/IEEE Design Automation Conference*, pp. 314-319, IEEE, 1994.